**Naval Research Laboratory**
**Stennis Space Center, MS 39529-5004**

NRL/MR/7441--95-7716

# The Conversion of Naval Search and Rescue Output into Vector Product Format

TOM FETTERER

*Planning Systems Incorporated*
*Slidell, LA*

KEVIN SHAW
GREG TERRIE
HILLARY MESICK
JONATHAN WRIGHT

*Mapping, Charting, and Geodesy Branch*
*Marine Geosciences Division*

H. VINCENT MILLER

*Mississippi State University*
*Stennis Space Center, MS*



May 17, 1996

# REPORT DOCUMENTATION PAGE

*Form Approved*
*OBM No. 0704-0188*



| 1. AGENCY USE ONLY *(Leave blank)* | 2. REPORT DATE | 3. REPORT TYPE AND DATES COVERED |
|---|---|---|
| | May 17, 1996 | Final |

**4. TITLE AND SUBTITLE**

The Conversion of Naval Search and Rescue Output into Vector Product Format

**5. FUNDING NUMBERS**

*Job Order No.* 574526700
*Program Element No.* RDT&E
*Project No.*
*Task No.*
*Accession No.* DN153251

**6. AUTHOR(S)**

Tom Fetterer*, Kevin Shaw, Greg Terrie, Hillary Mesick, Jonathan Wright, and H. Vincent Miller†

**7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**

Naval Research Laboratory
Marine Geosciences Division
Stennis Space Center, MS 39529-5004

**8. PERFORMING ORGANIZATION REPORT NUMBER**

NRL/MR/7441--95-7716

**9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)**

Naval Research Laboratory
Marine Geosciences Division
Stennis Space Center, MS 39529-5004

**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**

**11. SUPPLEMENTARY NOTES**

*Planning Systems Incorporated, 115 Christian Lane, Slidell, LA 70458
†Mississippi State University, Science and Technology Research Center, Stennis Space Center, MS 39529

**12a. DISTRIBUTION/AVAILABILITY STATEMENT**

Approved for public release; distribution unlimited.

**12b. DISTRIBUTION CODE**

**13. ABSTRACT** *(Maximum 200 words)*

The naval Digital Mapping, Charting, and Geodesy (MC&G) Analysis Program is involved in developing methods to combine Defense Mapping Agency (DMA) standard digital mapping products with other geographically oriented oceanographic data sets. These data sets include numerical model output and databases that reference attributes to specific geographic locations, but have not historically been considered cartographic products. The graphical representation of complex database and model output often speeds user understanding and interpretation of that data. In addition, the conversion of disparate data types to standard and supported formats reduces user training requirements, allows many users to derive benefits from a single application, and reduces program maintenance effort. This project demonstrates that modeling and simulation outputs can provide value-added overlays to standard digital mapping databases.

**14. SUBJECT TERMS**

digital MC&G, performance analysis, modeling and simulation

**15. NUMBER OF PAGES**

41

**16. PRICE CODE**

| 17. SECURITY CLASSIFICATION OF REPORT | 18. SECURITY CLASSIFICATION OF THIS PAGE | 19. SECURITY CLASSIFICATION OF ABSTRACT | 20. LIMITATION OF ABSTRACT |
|---|---|---|---|
| Unclassified | Unclassified | Unclassified | SAR |

CONTENTS

# THE CONVERSION OF NAVAL SEARCH AND RESCUE OUTPUT INTO VECTOR PRODUCT FORMAT

## 1.0 INTRODUCTION

The naval Digital Mapping, Charting, and Geodesy (MC&G) Analysis Program is involved in developing methods to combine Defense Mapping Agency (DMA) standard digital mapping products with other geographically oriented oceanographic data sets. These data sets include numerical model output and databases that reference attributes to specific geographic locations, but have not historically been considered cartographic products. The graphical representation of complex database and model output often speeds user understanding and interpretation of that data. In addition, the conversion of disparate data types to standard and supported formats reduces user training requirements, allows many users to derive benefits from a single application, and reduces program maintenance effort. This project demonstrates that modeling and simulation outputs can provide value-added overlays to standard digital mapping databases.

## 2.0 BACKGROUND

The objective was to prototype a technique of displaying oceanographic model output that is able to convey more information to the user. Using the Naval Search and Rescue (NAVSAR) model as input, a coverage in a DMA vector standard Vector Product Format (VPF) database was produced that could be overlaid on a Digital Nautical Chart (DNC). This product linked the model with DMA standard mapping software in a manner that can provide more useful information to a search coordinator than either product alone.

The technique was coded to offer a graphical user interface (GUI) to the conversion routines that monitored user input to reduce errors, thus saving time in critical situations. At present, the GUI does not offer all of the functionality that the SAR model could provide, but serves to demonstrate how models and databases can be made more useful by interacting with mapping software.

The building blocks of this effort were standard software and formats from the Navy and the Defense Mapping Agency, tied together with original code written in the C programming language. These elements consisted of the SAR model from the Oceanographic and Atmospheric Master Library (OAML) and the DMA's VPF spatial data standard.

### 2.1 Oceanographic and Atmospheric Master Library

OAML was established to provide meteorological, electromagnetic, oceanographic, and acoustic database and software support for the Fleet. It incorporates standardized and consistent environmental models and databases. The Naval Oceanographic Office (NAVOCEANO) is the project manager and provides software support activity for OAML.

OAML is designed to be a rapid-response, on-scene, environmental prediction software suite used to access the effects of the environment on the Fleet. Locally collected and archived meteorological information is used to prepare analyses of present atmospheric and electromagnetic conditions. The OAML suite can also provide best-guess predictions of environmental conditions using historical data from similar areas of interest.

## 2.2 Naval Search and Rescue

The NAVSAR model is a part of the OAML software library. It provides information and planning assistance during search and rescue incidents at sea. NAVSAR uses environmental data to compute the search object's drift, from time of distress, to determine the area in which the object will be at a later time. The probability function in the NAVSAR model divides the search area into multiple cells of equal size and computes the probability of the search object's presence in each individual cell. Other functions provide a search asset deployment plan, search asset on-station duration, and probability of overall search success estimates.

The NAVSAR model is intended to provide information and planning assistance to the Search Mission Coordinator during search and rescue operations. It also can provide assistance in the determination of the number of assets to commit to a search and how to maximize their effectiveness.

## 2.3 Vector Product Format

The DMA's VPF provides a standard relational format, structure, and organization for large geographic databases that are based on a georelational data model. VPF uses tables and indices that permit the direct access, by spatial location and thematic content, of any digital geographic data that can be represented using nodes, edges, and faces. Several mapping products are now available in VPF, such as the DNC and the World Vector Shoreline, with others in the planning and prototype stages.

In the VPF data model, the table is the organizational structure for all data content. A table contains three parts: a *header* (for metadata and column definitions), a *row id* (to indicate starting position of each row), and the *table contents* (for actual data, which is organized into rows and columns).

The five components of the VPF data model, from lowest to highest structural level, are Primitive, Feature, Coverage, Library, and Database (see Fig. 1). Primitives, taken together with additional thematic information, form a feature. Features are in turn grouped into *feature classes*. There are three types of feature classes: *simple, complex,* and *text*. The definitions of these types depend on the VPF tables that are present. At minimum, the feature class must contain at least one primitive table and precisely one feature table (this would be a simple feature class). A feature class schema table describes the rules for building feature classes by showing how each table should relate to other tables in the feature class.

Coverages, which encompass feature classes, consist of four mandatory components: *primitive directories* (named via tiling scheme), *value description tables*, *feature tables*, and the *feature class schema table*. A *data quality table* is optional. Tiling is the method of geographically subdividing a coverage for the ease of data storage. A library's tiling scheme is consistent among the coverages; however, any given coverage in the library may be untitled.

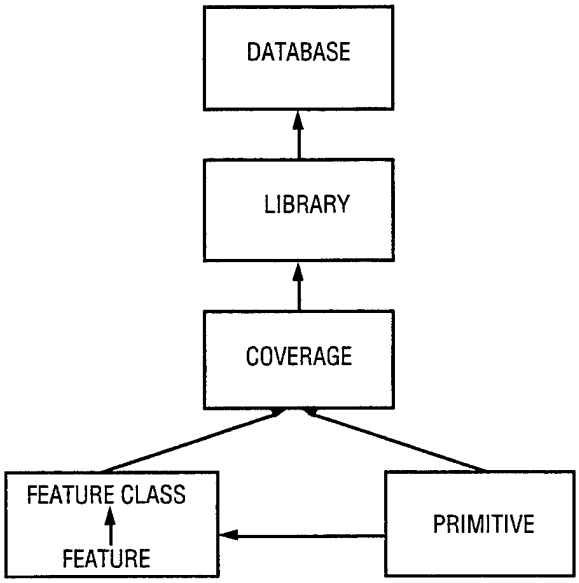


Fig. 1 — VPF Data Model Components

A library is a collection of coverages that share a single coordinate system and scale, have a common thematic definition, and are contained within a specified spatial extent (for example, four libraries of a DNC database may correspond to the General, Harbor, Approach, and Coastal paper charts of an area). Some sample coverages in a library are the Tile Reference Coverage and Library Reference Coverage (both mandatory if tiled coverages exist), Data Quality Reference Coverage (optional), and Names Placement Coverage. Some sample tables in a library are the Library Header Table and Geographic Reference Table.

A database is a collection of related libraries together with additional tables. The Library Attribute Table acts as a table of contents for the database. Database transmittal information is contained in the Database Header Table.

## 3.0 APPROACH

### 3.1 The NAVSAR Program Logical Structure

The program is divided into three logical blocks: the user input function, the SAR model, and the SAR to VPF conversion function. Each block is relatively self contained or, in the case of the SAR program, a separate program. The SAR model is interfaced by way of temporary files. Before the model is run, several files containing input parameters needed by the model are written. The model produces an output file containing a matrix of cell probabilities, which is read into the conversion routines. The temporary files are then discarded.

### 3.2 The User Input Functions

The user interface block gathers, error-checks, and formats the user data entered from the keyboard. This version is written to use X-Windows/Motif, but an MS-Windows version could be

substituted to allow the program to run on a PC. For a further description of the interface, see Appendix A.

### 3.3 The SAR Program

The SAR program is the programmatic implementation of the NAVSAR model. It is written in Fortran and is a part of the OAML library of environmental software. The model uses environmental data to compute the search object's drift from distress time to determine the most probable area to search. This is done by dividing the area into multiple cell areas of equal size and ranking them according to probability of search object containment. The model is capable of generating these probability cell grids using a bivariate normal distribution, for simple position scenarios, or Monte Carlo simulation for more complex trackline scenarios.

For information concerning the algorithms used in the model implementation, contact NAVOCEANO, Ocean Technology Division, or refer to the NAVSAR Software Design Document (OAML-SDD-35).

### 3.4 The NAVSAR to VPF Conversion Process

Once the SAR program has been run, a file containing a matrix of calculated cell probabilities (named pi4701.dat) exists. Before converting this data, the VPF database is checked to insure that no old NAVSAR data exists. If any existing coverages are detected, they are deleted. The NAVSAR model output file (pi4701.dat) is then read, checked for maximum and minimum geographical extents, and stored. These extents are compared with the extents in the database's library attribute table and a list of all libraries that contain NAVSAR points is produced. This step also checks to determine that the database has the proper read/write permissions to carry out the modification process.

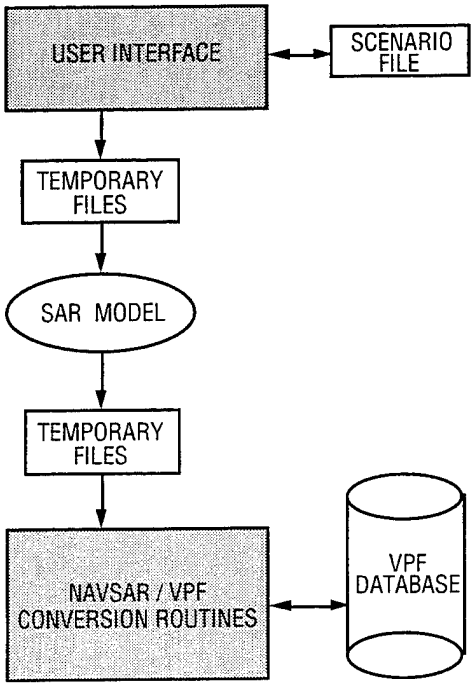


Fig. 2 — The NAVSAR Program Logical Flow

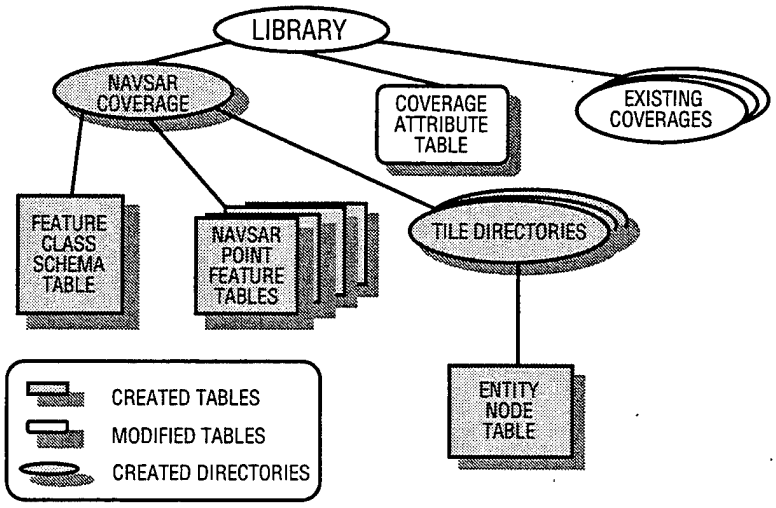


Fig. 3 — VPF Database Modifications

If any library in the database contains NAVSAR points, the point data's associated probability attributes are reviewed and a probability range array is determined. This is done by dividing the total range of probabilities into four equal ranges. The minimum probability is set to 0.0005 to avoid propagating any noise in the model output.

For each library identified earlier as containing NAVSAR points, each tile in that library is checked to determine whether any of these points lie within that tile. If so, the names and extents of that tile are stored for future reference. With this information, the stored NAVSAR points are sorted first by geographical position into a particular tile, and then by probability into one of the predetermined probability ranges. Once sorted, they are written to temporary files to be inserted into the database. Each of these temporary files represents a VPF point primitive table in the database. The temporary files are then written to the database after building the appropriate directory structure to reflect the tiles present.

To complete the coverage, a feature class schema table is written to the root node of this new directory structure. This table defines the relationship between the tables involved in the coverage. Essentially, it provides a logical map of the coverage's structure.

Finally, the coverage attribute table in each library is updated to reflect the addition of the NAVSAR coverage. This serves to register the coverage in the database and record where the files supporting it are located.

## 4.0 CONCLUSION

With the emerging standard formats for spatial data, such as VPF, digital mapping products are becoming more common in their availability and use. As this happens, it is helpful to try and incorporate older products, models, and databases into a format (VPF) that can be exploited by the newer standard mapping software. The integration of existing MC&G data sources into new or enhanced standard products both extends the useful life of the older products and makes the new products more useful.

The NAVSAR program demonstrates just such a marriage between older, but still valid codes, such as the SAR model and the new generation of digital mapping products. By a relatively simple conversion of data formats, both products are enhanced and their usefulness extended.

## 5.0 ACKNOWLEDGMENTS

This effort was sponsored by the Defense Mapping Agency, Mr. Bob Jacober serving as program manager, and the Defense Modeling and Simulation office.

Technical review of this report was provided by Mr. Mike Harris, Head of the Naval Research Laboratory Mapping, Charting, and Geodesy Branch.

# APPENDIX A

# USING THE NAVSAR PROGRAM

## INSTALLATION

The NAVSAR program may be run in any directory in which you have access permissions and that allows you to read and write temporary files. The program has an X-Windows resource file (named NAVSAR) that should be placed in a subdirectory named "app-defaults" below the NAVSAR program. This arrangement is the default configuration and the placement may be changed to another directory structure if you desire (see below).

Before running the program, you must set the environmental variable XFILESEARCHPATH to the path where the NAVSAR resource file is located.

example: setenv XFILESEARCHPATH /usr/navsar/%T/%N

The %T in this example is filled in by Motif as app-defaults, so this should be the name of the directory where the NAVSAR resource file is located. Likewise, the %N is replaced by the resource filename (NAVSAR). Both of these may be hard-coded to reflect the directory and filename if you wish to place the resource file elsewhere or name it differently.

example: setenv XFILESEARCHPATH /usr/navsar/defaults/Navsar_defaults

In the above example, the X-Windows resource manager is told to look for the resource filenamed "Navsar_defaults" in the */usr/navsar/defaults* directory.

Any VPF database you intend to modify with this program must have access permissions that will allow you to read and write directories as well as files. If this is not the case, the NAVSAR program will not function correctly. Any files created by this program will have world read/write permissions granted to them. This can only be changed by recompiling the program after changing the #define variable PERMISSIONS to the value desired. This constant can be found in the file nav.h.

example: #define PERMISSIONS 0700 /* Access granted to the user only */

## RUNNING THE PROGRAM

The program uses a scenario file to store all information relevant to a search problem. This file is a binary file and cannot be edited directly. The file scenario.dat is the default storage location for saved information, but another file may be chosen before a scenario is saved. This is covered in more detail below.

To start the program, type "Navsar" at the command line. This may be followed by a scenario filename if you want to change the default storage file, but this is optional.

Example: Navsar myScenario.dat

The program will initialize itself in one of the following ways:

1. If a command line scenario file designation exists and the file exists, it will open that file. NAVSAR scenario files are keyed with the NAVSAR program's version number and so, if the proper number is present, the program will load from that file. If the file key is improper, a default scenario will be created in its place.

2. If there is no command line argument present, the program will look for the last scenario saved in the default file scenario.dat and load the contents of this file.

3. If scenario.dat does not exist or has an improper version key number, a default scenario will be created.

## THE MAIN WINDOW EDIT FIELDS

The main window contains mandatory edit fields for date, time, position, and date and time of the map you want. These are simple text edit fields with the format provided in the default scenario. Leading zeros on the date/time and position fields are optional and serve only as a format reminder. At the top of the main window (and all other information popups) is a general user comment field that will hold up to 79 characters. It may be used for source or other miscellaneous information and is not used by the SAR model in any way.

*DTG (date/time group)* – This field should reflect the day, month, year, hour, and minute of the beginning of the search object's distress. All time in the NAVSAR program is expressed in Zulu time. DTG values are all mandatory except minutes, which may be entered as zero if not known.

*Last position (lat & lon)* – These two fields are the last known latitude and longitude of the search object. They are entered in degrees, minutes, and hemispheres (n/s, e/w). The hemisphere designation is not case sensitive and may be either upper or lower case letters.

*New DTG* – This field should reflect the day, month, year, hour, and minute of the map you wish to produce. Normally, this DTG would reflect the time (Zulu) at which you expect to be on the scene of the distress, but may be any time later than the original DTG.

*Navigational aid* – This is an option menu containing choices of navigational aids. This is the navigational aid used to determine the last position of the search object. Selection is made by first clicking on the box to open the menu and then clicking on the appropriate selection. If Inertial Navigation System is chosen, the number of hours since the last update will be requested. Selecting Cancel in response to this request will enter zero (just updated) into this parameter.

*Target type* – This is an option menu containing choices of search target types. The target type that best describes the object in distress should be chosen. It operates in the same manner as the navigational aid menu.

Below the target type menu is a push button that should be selected if the search problem involves a pilot who ejected from an aircraft. If selected (the push button is red), the SAR model takes into account the change of position caused by aircraft forward momentum and parachute drift. In this case, additional information is requested, including:

*Parachute deployment altitude* – This is the altitude (in feet) of the pilot when his parachute opened. This will normally be only conjecture, but is mandatory for a bail-out scenario. The program will not accept a zero in this field.

*Aircraft heading* – This is the heading of the aircraft at the time of pilot bail-out, measured in degrees clockwise from true north (not magnetic north).

## THE ENVIRONMENTAL AND WINDS ALOFT INFORMATION WINDOWS

Both the Environmental Conditions and Winds Aloft information windows have the same functions and work the same way. They are both called from the main window (the Winds Aloft window is only available in a bail-out scenario) and each contains four buttons at the bottom of the window:

*Apply* – Applies the information just entered to the current scenario. If you exit this window without saving the information with the apply button, it will still be there when you reopen the window, but is not yet a part of your scenario and will not be saved or offered to the SAR model. In addition, if either the *Next* or *New* button is pushed before you apply the information, that information will be lost. Existing information in any entry may be modified, but for it to take effect, it also must be applied.

*Close* – Closes the pop-up window.

*Next* – Switches between the applied entries.

*New* – Provides a new set of edit fields to fill out. As before, these entries must be applied before they are a part of the current scenario.

The Winds Aloft window may contain up to five wind readings. The measurements may be in any order convenient to you, as the model does not rely on order. The field at the top of each entry is a general user comment field that will hold up to 79 characters. It may be used for source or other miscellaneous information and is not used by the SAR model in any way. As a default message it simply records the sequence number of the entry.

The SAR model assumes that the wind is constant on both sides (top and bottom) of each measurement to a point midway between a given altitude and the next higher or lower measurement. The assumption is also made that the wind above the highest and below the lowest measurement is constant. Because of this, it is often helpful to make one of your wind entries a ground-level measurement (altitude = 0).

Each Winds Aloft reading consists of:

*Altitude of reading* – The altitude of the wind reading in feet.

*Wind speed* – The wind speed in knots.

*Wind direction* – The wind direction measured in degrees clockwise from true north.

The Environmental Conditions window may contain up to eight entries. The measurements may be in any order convenient to you, as the model does not rely on entry order. The field at the top of each entry is a general user comment field that will hold up to 79 characters. It may be used for source or other miscellaneous information and is not used by the SAR model in any way. As a default, it simply records the sequence number of the entry.

Inputs in this section describe the climatic conditions of the location where the distress occurred. Both observations from slightly before the time of distress to the present, as well as forecasts through the projected conclusion of the search, should be included.

Each Environmental Conditions entry consists of:

*DTG (date/time group)* – This field should reflect the day, month, year, hour, and minute of the entry (Zulu). All values are mandatory except minutes, which may be entered as zero if not known.

*Wind speed* – The surface wind speed in knots.

*Wind direction* – The surface wind direction measured in degrees clockwise from true north.

*Current speed* – The ocean current speed in knots.

*Current direction* – The ocean current direction measured in degrees clockwise from true north.

*Percent cloud cover* – A unitless percentage of cloud cover between zero (clear) and 100 (complete cloud cover).

*Visibility* – The visibility in miles of the search area.

*Oceanic mixed-layer depth* – The mixed-layer depth in feet. Usually gained from another environmental database.

## THE MAIN WINDOW FUNCTION BUTTONS

At the bottom of the main window there are three function buttons.

These buttons are:

*Quit Navsar* – Exits the program. The current scenario is *not* saved before the program terminates. This may be done before exiting by using the *Save Scenario* button.

*Save scenario* – Writes the current scenario to file. The program will prompt you for the scenario file to write to. At this time you may either choose OK to save the scenario in the file indicated, change the filename and then select OK (in which case the scenario is saved to the new filename), or choose Cancel to abort the operation altogether. If, after you change the filename in which to save your scenario, a file by that name exists, the program will warn you and ask for instructions. The choices are the same as the original ones above.

*Update database* – When this button is pushed, a dialog is opened to record the path to the VPF database you wish to modify. Clicking on Cancel will void your update command. If the OK button is pushed, the program performs the operations as described in the section *The NAVSAR to VPF Conversion Process*. When it is finished, the intermediate files are deleted and a list of VPF database libraries that have been modified is displayed. Each library listed contains at least one point in one of the four NAVSAR probability ranges.

## RESULTS OF A DATABASE UPDATE

When complete, the target database has been modified to reflect the probability of finding the target specified at particular points within each contained library. As mentioned, this modification is carried out as a point coverage grouped into four probability ranges. When displayed on a viewer, such as VPF_View, the coverage will look similar to Fig. 1. In this case, the red points indicate the area of greatest probability of finding the target. A spatial query on an individual point will show the rescue success probability at that particular point.

In this way a search area can be immediately identified, while more detailed information can be obtained as time permits. Additionally, other elements of the VPF database being used can be displayed alongside the NAVSAR point features to aid in identifying characteristics of the area in which the search is being conducted. This could include water depths, obstructions to navigation, and landmark points in the area. Such information can be invaluable to a search coordinator or field team.

## ERRORS AND TROUBLE SHOOTING

When an error occurs while running the NAVSAR program, a dialog box will appear with a general explanation of the problem. A more detailed explanation will usually be written to the program's parent window allowing you to remedy the situation. The most common reason for program failure is the lack of proper file access permissions on the database. It is recommended that the entire database have the same access permissions as the NAVSAR program. This version of NAVSAR assumes universal read/write permissions (777), and if you are having problems, your database should be changed accordingly. To do this, use the UNIX chmod command.

Example: chmod –R 777 /usr/dnc01

Where */usr/dnc01* is the VPF database path.

Appendix B

THE VPFTOOLS TOOLBOX

An outgrowth of the NAVSAR project has been a collection of routines for the manipulation of VPF tables. These functions allow the user to read, write, display, delete, locate, and copy elements or rows in a VPF table.

Written in C, these routines allow the programmer to concentrate on the functionality of his or her application without having to spend time learning the intricacies of VPF primitive operations. The toolbox is ANSI compliant and will compile under MS-DOS or UNIX.

This appendix contains a programmer's guide to the toolbox functions, as well as a listing of those functions and their calling parameters.

# Programmers Reference for the VPFTools Toolbox

v. 0.1

## Contents



This guide will help familiarize you with the VPFTools library of VPF table editing and display tools. These tools provide a simple way of accessing and manipulating VPF tables. This reference covers only level II functions. Level I functions (VPF primitive level) are described in the function reference section.

## General Concepts and Operations

There are several conventions to remember when using the VPFTools Toolbox. Data is always handled by the Toolbox functions as void pointers to VPF elements. An element, as it is used here, is a single instance of a VPF data type. They are defined as follows:

I – a single long integer (4 bytes)
S – a single short integer (2 bytes)
T – a single character (1 byte)
F – a single short floating point number
R – a single long floating point number
C – a pair (2) of short floating point numbers
B – a pair (2) of long floating point numbers
Z – a triplet (3) of short floating point numbers
Y – a triplet (3) of long floating point numbers
D – a date/time string (20 characters)
K – a triplet ID containing 3 long integer values
X – a null field containing no values

Except for the D, K, and X types, any of these element types may be contained in a variable-length field. As an example, a variable-length field containing three elements of the C data type would have six short floating point numbers in it. A field may also be fixed in length but contain multiple elements. The count parameter in the column definition will contain the number of elements found in a field or the word "variable" if that is the case. The symbolic constant VARIABLE may be used when specifying the count parameter in your programs.

The VPF standard treats triplet IDs as a special case. The data type of an individual ID is the smallest data type that the triplet value will fit into. For instance, the value 3 would be stored in 1 byte, whereas the value 259 could not be represented in one byte and would be stored as an short integer (2 bytes). To simplify matters somewhat, the Toolbox functions treat triplet IDs as an array of 3 long integers. The Toolbox functions actually read and write the data type whose size is appropriate to the value presented, but handle the values internally as long integers. This is what will be expected whenever a K data type is present.

Toolbox functions expect a VPF_TABLE structure to be complete, and the file in an open, read/write condition when it is presented. If this is not the case, serious problems may occur. The easiest way to insure the proper operation of the Toolbox routines is to build and/or open any VPF table with the *buildTable* or *loadTable* functions (see *Opening and Creating VPF Tables* below).

## Data Structures

Several data structures are used by the VPFTools Toolbox to hold information about an open table. A brief description follows about some of the more important ones.

The VPF_TABLE type is a structure that contains all the necessary information to manipulate a VPF table. This structure should only exist if a table is open. It is defined in tools.h as follows:

```
typedef struct {
  int table_type;          /* A table type number for a standard VPF table   */
  long int length;         /* The length of the table's header               */
  char *fileName;          /* The filename of the table                      */
  char *path;              /* Path to the directory in which that file resides */
  int openmode;            /* The read/write mode the file was opened in     */
  char modified;           /* A table content modified flag                  */
```

```
  char byte_order;          /* The byte order of the table – either L or M      */
  char *name;               /* The table's name                                 */
  char *narrative_file;     /* The table's associated narrative filename        */
  int num_columns;          /* The number of columns in the header              */
  long int num_rows;        /* The number of rows in the table                  */
  long int *var_index;      /* A pointer to a list of row lengths (variable index) */
  FILE *fptr;               /* The file pointer of the open table               */
  COLUMN_TYPE *header;      /* An array of COLUMN_TYPE structures               */
} VPF_TABLE;
```

The VPF_TABLE structure should be created and initialized using either *buildHeader* or *loadTable,* but if necessary, may be obtained by a call to *initializeTable* (see reference section). In this case, the table will be opened but not loaded, and the programmer will have to do this before it will be considered valid by any other Toolbox function.

The COLUMN_TYPE structure holds information that defines the table's header. It is normally created, populated, and placed in an array of such structures by the *buildColumn* function (see section *Building a Table* below). It is defined in tools.h as follows:

```
typedef struct {
  int field_length;         /* Number of VPF elements in the column             */
  char field_type;          /* The data type identifier (I, S, C etc.)          */
  char key_type;            /* The column's key type                            */
  char *name;               /* The name of the column                           */
  char *description;        /* A description of the column's contents           */
  char *value_description;  /* An optional value description filename           */
  char *thematic_index;     /* An optional thematic index filename              */
  char *narrative_file;     /* An optional narrative filename                   */
} COLUMN_TYPE;
```

Several Toolbox functions copy a portion or all of a table when inserting or deleting a row. These copy operations are carried out using a linked list of COPY_TYPE structure with a COPY_HEADER structure at the head of the list. The COPY_HEADER contains information on the type of copy operation done and records any swap files needed to accomplish the copy (more information is available in the *Utility Functions* section). The two structures are defined in tools.h as follows:

```
typedef struct {
  COPY_TYPE *first;         /* The head of the list of COPY_TYPE structures     */
  int mode;                 /* The storage mode of the copied material          */
  FILE *fptr;               /* A file pointer to a swap file if mode = DISK     */
  char filename[L_tmpnam];  /* The name of the swap file if used                */
} COPY_HEADER;

struct copy_node {
  char *row;                /* buffer containing the row's data                 */
  long int size;            /* The size in bytes of the row buffer              */
  struct copy_node *next;   /* A pointer to the next COPY_TYPE structure        */
};

typedef struct copy_node COPY_TYPE;
```

When building a row to insert into a table, it is often convenient to build a linked list of ROW_TYPE structures to hold your data. This is normally done using the *addElementToRow* function, which builds, replaces, and maintains the linked ROW_TYPE structures, but may also be handled by the programmer (see *ROW_TYPE List Handling Functions* below). Each node in the list corresponds to a column in the header of the table the row is being built for. The ROW_TYPE structure is defined in tools.h as follows:

```
struct row {
  void*element;          /* A void pointer to an array of VPF elements      */
  int count;             /* The number of elements in the element array     */
  struct row *next;      /* A pointer to the next ROW_TYPE structure        */
};
typedef struct row ROW_TYPE;
```

## Opening and Creating VPF Tables

*buildColumn*
*StandardHeader*
*buildHeader*
*buildStandardTable*
*loadTable*

This collection of Toolbox functions allows you to open existing VPF tables, create new standard tables, or create custom tables to suit your needs. Building custom tables requires a good deal of knowledge of the VPF standard and should be approached with respect. Opening and the construction of standard tables, on the other hand, is straightforward and painless.

The *loadTable* function requires only the filename of the table and an indication from you as to how the table should be opened.

*VPF_TABLE *loadTable(char *filename, int mode)*

The mode is the manner in which to attempt to open the table. This value must be one of the following integer constants defined in tools.h:

**EXISTING** – Read/Update – Will not create a new file if the one named is not found.

**READ_ONLY** – Opens the table in a read-only mode.

The default is EXISTING. The function returns a filled VPF_TABLE type structure containing the file pointer and other information needed to access the table data. If the file cannot be accessed, memory cannot be allocated for the VPF_TABLE structure, or the table is in some way invalid, NULL is returned.

The *buildHeader* function also opens a VPF table and returns a complete VPF_TABLE structure, but in this case, you need to supply all of the header information as well as the table name, byte order, and other information (see *Function* reference section). The building of an array of COLUMN_TYPE structures that define a header can be cumbersome, so there are two other ways to handle this situation.

If all of the information contained in the header is unique to your table, you will need to build a header from scratch. The easiest way to maintain a valid COLUMN_TYPE structure is to use the *buildColumn* function.

```
COLUMN_TYPE *buildColumn(COLUMN_TYPE *column,
                         int column_num, int operation,
                         char *name, char *ID, char type, char key,
                         int length, char *val_descr,
                         char *index, char *narrative)
```

The *column* argument is a pointer to the first node in the COLUMN_TYPE array or NULL if you are starting a new header. The *column_num* is the array position of the information in the following arguments. Columns are numbered zero through n with zero always being the Row_ID. Columns must be built in order beginning at zero. The *operation* argument can be one of the following:

**REPLACE_COLUMN** – The column denoted in *column_num* will be replaced in the header with the data in the following arguments.

**NEW_COLUMN** – The data in the following arguments will be appended to the header as a new column.

The rest of the arguments are described in the VPF standard and briefly outlined in the *function* reference section.

If the header information you are creating is similar to a standard table, but requires the replacement or addition of a column, the *standardHeader* function may be used to build a header from one of the standard tables in the Toolbox and simply replace or add columns as needed with the *buildColumn* function.

*COLUMN_TYPE *standardHeader(int type, long int *count)*

The *standardHeader* function requires only an integer type specifier and returns a pointer to an array of COLUMN_TYPE structures and fills the parameter *count* with the number of columns in the array. The table type specifiers are defined in tools.h as:

1 Connected node primitive table
2 Edge bounding rectangle table
3 Edge primitive table 4 Face primitive table
5 Face bounding rectangle table
6 Ring table
7 Feature class schema table
8 Feature class attribute table
9 Coverage attribute table
10 Library attribute table
11 Text primitive table

If you require a standard table, the function *buildStandardTable* may be used as a shortcut. This function builds and opens a standard VPF table of one of the above types and returns a pointer to a VPF_TABLE structure. The routine uses the above header-building functions to accomplish this.

Of great importance if you are building or modifying COLUMN_TYPE structures without using the Toolbox functions is the use of memory. COLUMN_TYPE structures and all of their component parts MUST be composed of dynamic memory allocated by malloc. The use of static or stack memory locations can cause serious problems, particularly for MS-DOS. This applies to the manipulation of these structures outside of the Toolbox functions only; arguments to these Toolbox functions themselves may be any type of memory.

## Reading from a VPF Table

Reading from a VPF table can be accomplished in two ways: the *getRow* function and a direct-read using the level I primitive function *readElement*. The *getRow* function reads an entire row and stores it in a ROW_TYPE structure for which a pointer is returned. More information on ROW_TYPE structure handling functions can be found in the section *ROW_TYPE List Handling Functions.*

The *readElement* function returns a pointer to an array of elements found at the current file pointer position and the number of elements in the array.

*void *readElement(VPF_TABLE *table, int index, int *count)*

The argument *table* is the table you are working with, *index* is the column that corresponds to the column the data you wish to read is in, and *count* is a pointer to an integer that *readElement* fills in with the element count of the returned array. To read an element from a VPF table with *readElement*, the file pointer to that table must first be set to the position of that element in the file. This can be done with the *setCurrentPosition* function (see *Utility Functions*). The returned array should be cast to the proper data type before use and, as it is dynamically allocated memory, be freed when you are through with it.

## Writing to a VPF Table

*WRITEROW*
*writeRowFromList*

There are two functions available for the modification of VPF tables: *writeRow* and *writeRowFromList*. Both have similar arguments and their results follow the same pattern. Both also maintain any variable-length indices associated with a table; an operation that can be cumbersome to do manually.

SYNTAX:

*long int writeRow(VPF_TABLE *table, int function, char *fmat, ...)*
*long int writeRowFromList(VPF_TABLE *table, int function, ROW_TYPE *list)*

The first argument in both functions is the VPF table in which to write. This must always be a pointer to a valid VPF_TABLE type structure initialized by either the *loadTable* or the *buildHeader* functions described above. The file must have been opened in a mode other then read only. Your "write" will fail if write permission is not granted on the file you wish to modify.

The second argument is the write function you wish to perform. Possible choices have constants defined in tools.h and include:

**INSERT_ROW** – Inserts your new row at the position specified in the Row_ID column of your row. Any existing rows in the table are renumbered to reflect the addition of the new row. If, for some reason, the row number provided is invalid, your row will be appended to the end of the table. By using the symbolic constant END_ROW (defined in tools.h) as your row designation, the write function will revert to the default APPEND_ROW.

**APPEND_ROW** – Appends the row you wish to write to the end of the table. When this constant is specified, any other row designation is ignored and the row is always placed at the end of the table. This is convenient if you do not wish to look up the number of the last row of a table. This operation is also faster than INSERT_ROW because no check is made to see if any rows exist after the point where this one will be placed.

**REPLACE_ROW** – Replaces the row at the position specified in the Row_ID column of your row. The existing row is overwritten. Again, this write function will default to APPEND_ROW if the row specification is invalid.

From here the two functions specialize. The function *writeRow* should be used when both the number of arguments and their data types are known in advance, for instance, a routine that writes a certain type of table and only that table. It provides a variable-length argument list so that this one function may be used to write to any table. To keep the function compatible with MS-DOS compilers, the argument *fmat* is included. It is a pointer to a string containing the format of the first argument in the variable argument list. Because the first argument is always the Row_ID, this string should always be specified as: *char *fmat = "%ld";*.

The variable-length argument list follows and observes the following conventions:

1. The first argument is ALWAYS the row number where you wish the row to be written. This must be a long integer.

2. The remainder of the arguments are pointers to the data in your row in the order of the columns in the table's header. In the case of coordinate data types or multiple counts (field lengths), the pointer will be to the first value in an array of values. If a column contains a variable-length field, an integer (not a pointer to an integer) denoting the number of elements in the following argument MUST precede the pointer to the array.

The function will read each argument according to the corresponding data type in the table's header, so care must be taken to order them correctly. In the case of a null data element (VPF data type X), no argument should appear in the variable argument list of the function call.

The *writeRowFromList* function should be used when you don't know in advance the data types to be written. A routine that handles a variety of different table types is an example. The final argument in this function is a pointer to a linked list of ROW_TYPE structures (see *Data Structures* above). The list is processed by *writeRowFromList* in the order of the table's header with each node in the list corresponding to a column in the header. The list presented must be complete or an error will occur. There are several functions to build and maintain these ROW_TYPE lists outlined in the next section. In the case of a null data element (VPF data type X), no entry need be made in the list, for the *addElementToRow* treats all nonspecified row information as null pointers.

Both *writeRow* and *writeRowFromList* return the same values. If successful (the row was written to the table), the row number at which the row was written is returned. This may not be the row number you asked the function to write to if the specified row was not valid. "On error a –1" is returned. The primitive level write functions used by these functions do not check whether the write to file was successful, so if a problem such as lack of write permissions exists, a successful write operation may be flagged when, in fact, the write failed. Access problems such as these should be checked before any write operation is initiated (see the *getAccess* function in the *Utility Functions* section below).

**Program example:** In this example we assume the VPF table *txt.* exists but contains no rows. Memory is allocated for the coordinate pair and the text string. These are filled and *writeRow* is called. Because both columns 1 and 2 are variable-length fields, the number of elements in each of those arrays is given as part of the argument list. A loop is then entered in which both the coordinates and text string are changed and a new row is written with each iteration. The table is then closed before exiting the program.

```
#include "tools.h"

void main(void)
{
        VPF_TABLE *table;
        ROW_TYPE *row = NULL;
        char *fmat = "%ld", *string;
        float *coord;
        int i;
        if((table = loadTable("txt.", EXISTING)) == NULL)
        {
                printf("can't open table\n");
                exit(-1);
        }
        coord = (float *)malloc(sizeof(float) * 2);
        coord[0] = -76.040000;
        coord[1] = 36.880335;
        string = (char *)malloc(14);
        sprintf(string,"This is row 1");
        writeRow(table, APPEND_ROW, fmat, 1L, strlen(string), string, 1, coord);
        for(i = 2; i < 4; i++
        {
                sprintf(string,"This is row %d", i);
                row = addElementToRow(row, (void *)string, strlen(string), 1);
                if(i == 2)
                {
                        coord[0] = -76.000000;
                        coord[1] = 36.880000;
                }
                else
                {
                        coord[0] = -76.090000;
                        coord[1] = 35.665430;
                }
                row = addElementToRow(row, (void *)coord, 1, 2);
                writeRowFromList(table, APPEND_ROW, row);
```

```
            row = freeRow(row);
            coord = (float *)malloc(sizeof(float) * 2);
            string = (char *)malloc(14);
        }
        closeTable(table);
}
```

## ROW_TYPE List Handling Functions

*addElementToRow*
*getRow*
*writeRowFromList*
*freeRow*

The ROW_TYPE list allows the programmer to construct a row from any type of table using the same structure. It is a linked list of ROW_TYPE structures (see *Data Structures* above) in that each holds the data from one of the columns of the row being built. Several Toolbox functions support list creation, usage, and destruction, making the maintenance of these structures relatively easy.

To create a row, two options are available: *addElementToRow* allows you to create a row from scratch and *getRow* retrieves a row from a VPF table.

*ROW_TYPE *addElementToRow(ROW_TYPE *start,*
*void *element, int count, int position)*

This function takes as arguments a pointer to the first node in a list (**start*), the element array to place in the list, the number of elements in that array, and the column position at which to place the data. If the argument **start* is NULL, a new list is begun and a pointer to it is returned. Each node in this list corresponds to a column in the table, so if you should specify the position you want an element placed in a newly created list as 3, nodes 0, 1, 2, and 3 will be added to your list and the element will be placed in node 3. Nodes 1 and 2 will contain NULL values. The first node (node or column zero – the row ID) is a special case where if you do not specify a Row_ID, this column is given a default of END_ROW or the end of the list. No other column has a default, and presenting a non-complete row to another of the list-handling functions can have unpredictable results.

Values may be changed in a list simply by calling *addElementToRow* with the new value. The old value will be freed and the new one installed in its place. It is recommended that any element array used in a list function be dynamically allocated memory. The use of static memory may cause problems, particularly in the *freeRow* function. The use of stack storage locations can cause segmentation faults.

*ROW_TYPE *getRow(VPF_TABLE *table, long int row)*

The alternate way of getting a row is to use *getRow* to obtain a copy of a row in the applicable table in a ROW-TYPE list. This list may then be modified as needed and inserted in a new position in the table. This works particularly well for building tables whose columns from row to row are, for the most part, identical. For instance, an entity node primitive table where only the point coordinates change could be retrieved from an *end.* table, the coordinates changed using

*addElementToRow,* and the row written to the end of the table using *writeRowFromList* (see example). The *getRow* function can also be used to retrieve or change individual elements from a table.

Once a row has been created and filled, *writeRowFromList* can be used to insert the row into a table (see *Writing to a VPF Table* above).

After you are through with a row, *freeRow* may be used to free the memory allocated for the row.

**Program example**: In this example, the VPF table *end.* is opened and the first row is retrieved from it. Memory is then allocated for the new coordinate pair (two floating point numbers) and the new coordinates are stored into the space. The new element is then added to column 3 (the COORDINATE column) and the modified row is written to the end of the table. Cleanup is performed by the *freeRow* function, which deallocates the row's memory, and the table is closed before exiting the program.

```
#include "tools.h"

void main(void)
{
        VPF_TABLE *table;
        ROW_TYPE *row;
        float *element;

        if((table = loadTable("end.")) == NULL)
        {
                printf("can't open table\n");
                exit(-1);
        }
        row = getRow(table, 1L);
        element = (float *)malloc(sizeof(float) * 2);
        element[0] = -76.040000;
        element[1] = 36.880335;
        row = addElementToRow(row, (void *)element, 1, 3);
        writeRowFromList(table, APPEND_ROW, row);
        freeRow(row);
        closeTable(table);
}
```

## Utility Functions

*copyRow*
*closeTable*
*copyTable*
*rewriteTable*
*Delete*
*find*
*setCurrentPosition*
*writeTableDataToFile*
*readDataFile*

The Toolbox offers many utility functions which, for the most part, are self explanatory. A few notes on some of the more important ones are in order. For a more complete listing, see the *Function* reference section.

The *closeTable* function closes a table and deallocates the memory used in the various structures. Always use *closeTable* to close all open tables before exiting a program. This will insure that any variable-length indices are updated.

The *copyRow* function is a convenience function that simply copies a row from one position in a table to another. This routine uses the write functions mentioned above, and may append the row if you specify a nonexistent destination row. If in doubt of the existence of a particular row, use the *validQuery* function to determine if it exists.

The functions in the Toolbox that reorder VPF tables (*delete*, *write*, etc.) use the *copyTable* and *rewriteTable* functions to record the existing rows before overwriting the table. The *copyTable* routine copies a table from a specified row to the end of the table and returns a COPY_HEADER structure that holds a linked list of the table's rows. If memory runs out while copying the table (or memory falls below 5K on a DOS machine), the rows are swapped to a temporary file and the copy continues. In such cases when the header is returned, it contains the first row copied in a linked COPY_TYPE structure and all subsequent rows in the swap file whose file pointer is in the header structure's *fptr* field. The name of the file is also recorded in the *filename* field. The structure's *mode* field will contain either the integer constant DYNAMIC, which indicates that the copy was all done in memory, or DISK, which means the swap file option was used.

If you use *copyTable* to move data about, it's easiest to also use *rewriteTable*, which does whatever is necessary to replace the data without any programmer attention. It is also advisable to use *freeCopyBuffer* to deallocate the COPY_HEADER structure once you are through with it.

The *Delete* function simply deletes a row from a table. The *find* function searches a table for an element or group of elements in a single field. The values to search for must be entered exactly as they appear in the table. Remember, an element may be a 2- or 3-value coordinate set or a single value, depending on the column's data type. This function compares elements to find a match. No wild cards are supported at present. The function will return the row at which the key entry was found. When working with variable-length fields, only the number of elements you entered as keys will be compared. If the table's field has 4 elements and you have entered the first 3 correctly, that row is flagged as a match. If, however, you have entered correctly the 4 elements plus 1 additional element, the function would not find a match.

The *setCurrentPosition* function allows you to move the table's file pointer to whatever row column position you want. The function requires the table you are working on and the row/column combination to move to. It places the file pointer at the beginning of an element field (which may not be the element itself in the case of a variable-length field) and returns the byte offset of the position it has placed the pointer. If an invalid row/column combination is presented to it, *setCurrentPosition* will return a –1.

The *writeTableDataToFile* function writes the data from a VPF table into an ASCII file. The *readDataFile* reads an ASCII data file and writes that data into a VPF table. The files produced by *writeTableDataToFile* and read by *readDataFile* are formatted in the same way: The data should be <CR> delineated ASCII text with any variable-length field preceded by an integer denoting the number of elements in that field. Again, this number is elements, not numbers or letters.

**Example of an ASCII data file**: This data file is to be written to a text primitive table (txt) which contains 3 columns: ROW_ID, STRING, and SHAPE_LINE. There are two rows present:

```
1                       /* Row_ID                                                  */
9                       /* Number of elements in the following variable-length field */
Annapolis               /* The text string                                         */
2                       /* Number of elements in the following variable-length field */
-76.188823 36.667234    /* The shape coordinates                                   */
-76.188844 36.667233
2                       /* Row_ID                                                  */
8                       /* Number of elements in the following variable-length field */
Cape May                /* The text string                                         */
1                       /* Number of elements in the following variable-length field */
-76.233823 36.667934    /* The shape coordinates                                   */
```

The comments are for explanation only and should not appear in this file. The formatting is forgiving, these entries could all be on one line and it would still work. If using fixed-length text fields, however, in your data file, you must either allow the field width specified in the *field_length*; parameter of that column or delimit the end of the string with a <CR>.

## Creating Supporting Tables

*createEBRtable*
*createFBRtable*
*writeVarIndexTable*

Two supporting bounding rectangle table functions are offered in this version of the Toolbox: the *createEBRtable*, which builds an edge table minimum bounding rectangle table, and the *createFBRtable*, which builds a face table minimum bounding rectangle table. The *createEBRtable* function is straightforward and should be run after all modifications to the edge table are complete. It will replace any existing EBRtable with an updated one.

The *createFBRtable* function should also be used after all other face supporting tables are complete and closed. This includes the ring table, edge table, and the connected node table. It is especially important that the edge table used to store the face definitions be closed prior to the running of the *createFBRtable* routine as this routine requires an updated variable-length index, which is only written at table closure.

The *writeVarIndexTable* function is normally not called by the programmer, but rather, is called by the *closeTable* function when the table is complete. This table will write the variable-length index table required by VPF tables with variable-length fields. There is no restriction on when the function may be called, however, so if a situation arises where the table you are working with needs to be completely updated, but remain open, it may be called.

## Function Reference Section

*Levels I and II*

A listing of VPFTools functions.

All functions require the include file tools.h.

---

**addElementToRow** – Creates and maintains a ROW_TYPE structure. Will build, add an element to, or replace an element in a linked ROW_TYPE structure. Does not validate the structure and will place an element at any position in the list you specify whether or not that position is valid to the table. The exception to this is the first node, which contains the mandatory row number. This value is set to the constant END_LIST (number of rows +1) if you do not set it to another value. If you wish to create a new list, the first call to this function should contain a NULL value for the start parameter. Returns the head of the list.

SYNTAX: ROW_TYPE *addElementToRow(ROW_TYPE *start,
void *element, int count, int position)

ARGUMENTS:
INPUT:

*start* – A pointer to the head of a valid ROW_TYPE structure or NULL if you wish to start a new row.

**element* – A void pointer to a VPF data element (or array of elements). This must be contained in memory allocated by malloc.

*count* – The number of elements contained in *element.*

*position* – The column in the row in which the data presented belongs with the first position (the mandatory row identifier) being column zero.

Output: The head of the list in which you just inserted your data or a NULL pointer if an error occurred.

---

**buildColumn** – Properly fills and configures a COLUMN_TYPE structure in the header array. *buildColumn* allows you to replace a column or append one to a table header depending on the argument operation. A NULL pointer can be passed to *buildColumn* to begin a header array and then the returned pointer passed to subsequent calls to *buildColumn* to continue building the header.

SYNTAX: COLUMN_TYPE*buildColumn(COLUMN_TYPE *column, int
column_num, int operation,

char *name, char *ID, char type, char key,
int length, char *val_descr,
char *index, char *narrative)

ARGUMENTS:
INPUT:

*column* – A pointer to a COLUMN_TYPE structure containing the definition of a column of a VPF table.

*column_num* – The number of the column (zero - n) where column zero is the row number and n is the last column in the table.

*operation* – An integer constant denoting the operation to perform on the header. The options as defined in tools.h are:

REPLACE_COLUMN – The column *column_num* will be replaced in the header with the data in the following arguments.

NEW_COLUMN – The data in the following arguments will be appended to the header as a new column.

The rest of the arguments mirror the data stored in a column of a VPF table and are outlined briefly here:

*name* – (char *) column name (n < 17)
*ID* – (char *) column textual description
*type* – (char) a field designator from table 11 MIL-STD-2407 (VPF spec.)
*key* – (char) a key designator from table 12 MIL-STD-2407 (VPF spec.)
*length* – (long) length of the column's data
*val_descr* – (char *) optional value description table name
*index* – (char *) optional thematic index filename
*narrative* – (char *) optional column narrative filename

Output: Returns a pointer to an array of COLUMN_TYPE structures containing the header description or a NULL on error.

---

**buildHeader** – Constructs a VPF header from information provided in a VPF_TABLE structure as well as other parameters. There is no error checking done, on this structure so the resulting table is only as good as the values stored there. This function will leave the file pointer at the end of the newly created table header.

SYNTAX: VPF_TABLE *buildHeader(COLUMN_TYPE *column, int num_columns,
char byte_order, char *table_description,
char *narrative, char *filename)

ARGUMENTS:
INPUT:

*column* – An array of COLUMN_TYPE structures containing the following:

*name* – (char *) column name (n < 17)

*field_type* – (char) a field designator from table 11 MIL-STD-2407 (VPF spec.)

*field_length* – (int) length of the column's data

*key_type* – (char) a key designator from table 12 MIL-STD-2407 (VPF spec.)

*description* – (char *) column textual description

*value_description* – (char *) optional value description table name

*thematic_index* – (char *) optional thematic index filename

*narrative_file* – (char *) optional column narrative filename

Note that all optional fields, if they are not being used, must contain NULL. If they are used, the entries in them must be < 13 characters. All other fields are required and must be specified.

*num_columns* – An integer specifying how many columns are in the table.

*byte_order* – A character representing the byte storage order used. This must be either "M" or "L" as defined in section MIL-STD-2407, section 5.4.1.1.

*table_description* – A character string describing the table's contents. It must contain 80 characters or less.

*narrative* – An optional narrative filename of < 13 characters. If it is not used, this parameter must be NULL.

*filename* – A pointer to the filename of this table including any necessary path.

Output: A completed VPF_TABLE structure with all necessary fields filled in. Will return NULL Possible errors are a file that could not be opened or the lack of memory to allocate.

---

**buildStandardTable** – Creates a standard VPF table. These tables were all taken from the DNC01 database but meet the VPF standard. Returns a pointer to a filled VPF_TABLE structure.

SYNTAX: VPF_TABLE *build StandardTable (int type, char *filename, char byte_order)

ARGUMENTS:
INPUT:

*type* – An integer constant denoting the type of header to create. A list of the available standard table types and their constants can be found in tools.h.

*filename* – A pointer to the path/filename of the table you wish to create.

*byte_order* – A character constant denoting the byte storage order of the table you wish to create. It must be either "M" or "L" as defined in MIL-STD-2407, section 5.4.1.1.

Output: Returns a pointer to a VPF_TABLE structure containing a new instance of the table specified by *type*. On error, NULL will be returned.

---

**closeTable** – Closes and deallocates memory associated with a VPF_TABLE type structure. Also updates the variable-length index file. Uses two functions to free allocated memory – *freeTable* and *freeRow*. Returns NULL.

SYNTAX: VPF_TABLE *closeTable (VPF_TABLE *table)

---

**copyRow** – Copies a row from one position to another in a VPF table. This function will either overwrite data in the destination row or insert it, depending on the value in *function*.

SYNTAX: long copyRow(VPF_TABLE *table, long from, long to, int function)

ARGUMENTS:
INPUT:

*table* – A pointer to a valid VPF_TABLE type structure.

*from* – The number of the row you wish to be the source of the copy.

*to* – The number of the row you wish to be the destination of the copy.

*function* – The write function to perform. This is an integer constant defined in tools.h, which can be either:

INSERT_ROW – Writes the row to the row number specified and the remaining rows are reordered to accommodate the new row. No data is lost in this operation and any applicable variable-length index is updated.

REPLACE_ROW – Replaces an existing row with the data from the source row. The existing data at this row is lost.

Output: Returns the number of the row at which the source data was actually written or a –1 if the source row specification was invalid.

---

**copyTable** – Copies the contents of a table from "row" to the end of the table into a linked list. Returns the head of that list. If the system should run short of memory, the table is swapped to a temporary file in the directory specified by the environmental variable TMP. If this variable is not set, TEMP is checked. If this, too, is not set (or you do not have write permission in these directories), the file is placed in your current working directory.

SYNTAX: COPY_HEADER *copyTable(VPF_TABLE *table, long int row)

ARGUMENTS:
INPUT:

*table* – A pointer to a valid VPF_TABLE type structure from the table you wish rows copied from.

*row* – The row you wish to start with when copying. The last row in the table is the last one copied.

Output: Returns a pointer to a COPY_HEADER type structure in which the copy buffer is detailed or NULL on error. Possible causes of an error condition are:

1. Lack of memory for the copy structure's header
2. Specification of a row not in the table
3. Lack of memory for the copy buffer (the size of the first row)
4. Unable to open the temporary swap file
5. Unable to free up enough memory after the swap

---

**createEBRtable** – Creates a minimum bounding rectangle table from data contained in the argument *table*. The table passed must be an edge primitive table. The EBRtable will be given the standard name ERB and will be written to the directory where *table* is located.

SYNTAX: int create EBRtable(VPF_TABLE *table)

ARGUMENTS:
INPUT:

*table* – A valid VPF_TABLE structure with the TABLE_TYPE field set to EDG.

Output: Returns a zero if the table was successfully created or a –1 if the table passed was the wrong type or the EBR table could not be created.

---

**createFBRtable** – Creates a minimum bounding rectangle table from data contained in the argument *table*. The table passed must be a face primitive table. The FBR table will be given the standard name FBR and will be written to the director where *table* is located. To create an FBR table, an edge bounding rectangle table (EBR), edge primitive table (EDG), and a ring table (RNG) must have already been created and filled.

SYNTAX: int create FBRtable(VPF_TABLE *table)

ARGUMENTS:
INPUT:

*table* – A valid VPF_TABLE structure with the TABLE_TYPE field set to FAC.

Output: Returns a zero if the table was successfully created or a negative number if an error occurred. Error codes returned may be:

–1 An error in one of the files used to build the FBRtable was detected. In this case, the FBRtable may be partially filled but will be a valid table.
–2 The table passed was the wrong type or the FBR table could not be opened for some reason.
–3 The edge primitive EDG table could not be accessed.
–4 The ring (RNG) table could not be accessed.
–5 The edge bounding rectangle (EBR) table could not be accessed.
–6 Memory could not be allocated for needed structures.

---

**Delete** – Removes a specified row from a VPF table. Returns the number of the row deleted if successful, zero if the row specified is not found and –1 if an error occurs. Any variable-length index associated with the table will be modified to reflect the deletion.

SYNTAX: int Delete(VPF_TABLE *table, long int row)

ARGUMENTS:
INPUT:

*table* – A pointer to a VPF_TABLE type structure from the table you are working on.

*row* – The row you want to delete.

OUTPUT: An integer error code where:

0 = entry not found

>0 = the row number where the row was deleted

>0 = an error:

–1 = the specified file could not be accessed for some reason or the *row* argument was invalid.

---

**The dump utilities:**

These are utility functions that simply dump a VPF table and its data to the screen.

*dumpTable* – A convenience function that dumps both the table header and the table data.

SYNTAX: void dumpTable(VPF_TABLE *table)

*dumpTableData* – Dumps the data portion of a VPF table.

SYNTAX: void dump TableData(VPF_TABLE *table)

*dumpHeader* – Dumps a VPF header contained in a VPF_TABLE structure.

SYNTAX: void dump HEADER(VPF_TABLE *table)

*dumpRow* – Dumps a specified row from a VPF table.

SYNTAX: void dumpRow(VPF_TABLE *table, long int row)

*dumpElement* – Dumps an element from a specified row and column in a table.

SYNTAX: void dumpElement(VPF_TABLE *table, long int row, int column)

---

**elementSize** – Returns the size of an individual VPF type element.

SYNTAX: long int elementSize(COLUMN_TYPE *format)

PARAMETERS:
INPUT:

*format* – The column in the VPF_TABLE that contains the format of the element you choose the size of. This corresponds to the type of the element, but is an integer denoting the COLUMN_TYPE structure that holds that information. If the column is a variable-length field, *elementSize* will return the size of one element of this data type. Use the *get VariableFieldSize* function to get variable-length field sizes.

Output: Returns the size in bytes of the element.

---

**find** – Returns an array of row numbers at which the data element passed as *elements* was found. The end of the list of row numbers is marked by a node containing zero.

SYNTAX: long int *find(VPF_TABLE *table, void *elements, int count, int column)

ARGUMENTS:
INPUT:

*table* – A pointer to a VPF_TABLE type structure from the table you are working on.

*elements* – A pointer to a single or multiple data elements to compare. If the *fieldType* is T, *elements* would be a pointer to a character string; if *fieldType* is C, *elements* would be a pointer to an array of 2 floating point values containing the lat and lon of the point to find, etc.

*count* – The number of elements in the argument *elements* above.

*column* – The column (and hence data type) of the values in *elements*.

Output: A pointer to an array of long integers where:

array[0] = 0 – entry not found
array[0] – array[n], the row numbers where the entry was found
NULL = An error such as the specified file could not be accessed for some reason, or the *column* argument was invalid.

---

**findColumn** – Compares the argument *name* to each column name in a VPF table header and returns the number of that column if found. If not found it returns a –1. This is a simple case-insensitive string compare so the argument passed as *name* must be an exact match.

SYNTAX: int findColumn(VPF_TABLE *table, char*name)

ARGUMENTS:
INPUT:

*table* – A pointer to a VPF_TABLE type structure from the table you are working on.

*name* – A character string containing the field ID name of the column you wish to locate.

Output: Returns an integer denoting the column ID that matches the argument *name*, or –1 on error.

---

**formatDate** – Returns a string with the date string from a VPF table decoded and formatted for ease of reading. The format of the input string must conform to the VPF standard (MIL-STD-2407). The date string returned is contained in memory allocated with malloc and should be freed when no longer needed.

SYNTAX: char *formatDate(char *date)

ARGUMENTS:
INPUT:

**date* – A pointer to a 40-character string containing the coded date/time information from a VPF table.

Output: Returns a pointer to a string containing the date and time in plain English.

---

**freeColumn** – Deallocates dynamic memory allocated for a COLUMN_TYPE structure. The argument *column* is a pointer to just such a structure.

SYNTAX: void freeColumn(COLUMN_TYPE *column)

---

**freeCopyBuffer** – Cleans up after a copy by closing any temp files and freeing the linked list of copy structures.

SYNTAX: void freeCopyBuffer(COPY_HEADER *header)

ARGUMENTS:
INPUT:

*header* – A pointer to a valid COPY_HEADER type structure created by *copyTable.*

Output: None.

---

**freeTable** – Frees dynamic memory allocated for a VPF_TABLE type structure. Returns NULL.

SYNTAX: void freeTable(VPF_TABLE *table)

---

**getAccess** – Returns the highest access code available for the file presented in the argument *filename.* Possible return values are:

6 – Read and write access available
4 – Only read access available

2 – Only write access available
0 – File does not exist or has execute only permission

SYNTAX: int getAccess(char *filename)

---

**getData** – Returns a binary representation of data from an ASCII representation in parameter *string*. Will set a field to zero if the string presented contains invalid data (or a NULL string in the case of T or D data types). Returns allocated dynamic memory which should be freed when you are through.

SYNTAX: void *getData(VPF_TABLE *table, char *string, int column, int *count)

ARGUMENTS:
INPUT:

*table* – A pointer to a valid VPF_TABLE type structure.

**string* – A pointer to a character string containing the data to be converted.

*column* – The column in the VPF_TABLE structure's header field that contains the format or element you wish to convert.

*count* – The number of elements contained in *string*.

Output: Returns a void pointer to an array of data elements formatted in the data type described in *column[count]*. On error, NULL is returned. An error is not generated by invalid data contained in *string*. In the case of invalid numeric data, an array will be returned containing zeros in place of the bad data.

---

**getRow** – Reads a specified row from a VPF table into a ROW_TYPE structure.

SYNTAX: ROW_TYPE *getRow(VPF_TABLE *table, long int row)

ARGUMENTS:
INPUT:

*table* – A pointer to a valid VPF_TABLE type structure.

*row* – The number of the row you wish to read.

Output: The head of a ROW_TYPE structure containing the data from the row specified. If an error occurs a NULL pointer is returned.

---

**getVariableFieldLength** – Gets the field length in bytes of any field in a VPF table. Returns –1 if you give it a nonexistent row/column combination. It will return any field length, not just variable-length ones. This function supersedes the *elementSize* function, though that function may still be used if needed.

SYNTAX: long int getVariableFieldLength(VPF_TABLE *table, long int row, int column)

---

**grabHeader** – Dumps a VPF_TABLE structure into a file in a case format to be used in the *buildStandardTable* and *standardHeader* functions below. This routine was written as a development tool and probably has limited application elsewhere. If you need to extend the standard table set in function *buildStandardTable*, however, this will provide a handy way of formatting your data. The output written to file contains first, the lines to be cut into *standardHeader* and, below this, the line for *buildStandardTable*.

SYNTAX: void grabHeader(VPF_TABLE *table, FILE *fptr, char *switch_name)

ARGUMENTS:
INPUT:

*table* – A pointer to a VPF_TABLE type structure from the table you are working on.

*fptr* – A pointer to an open file to which you want to write the data.

*switch_name* – The symbolic constant to be placed after the *case* key word that refers to the table information to be written.

Output: No direct output – writes header data to file pointed to by *fptr*.

---

**initializeTable** – Allocates and initializes a VPF_TABLE type structure. Also opens the file named by *filename*. Returns NULL if unsuccessful. This function is normally not called directly, but by the *loadTable* or *buildTable* functions.

SYNTAX: VPF_TABLE *initializeTable(char *filename, int mode)

PARAMETERS:
INPUT:

*filename* – A pointer to the filename of a VPF table.

*mode* – An integer constant denoting how a particular file should be opened. The three choices defined in tools.h are:

*EXISTING* – Read/update – Will not create a new file if the one named is not found.

*CREATE* – Write/update – Will open a new file with the filename given if one does not exist. The default is *EXISTING*.

*READ_ONLY* – Opens the table in a read only mode.

Output: Returns a partially filled VPF_TABLE structure.

---

**loadTable** – Opens the VPF file contained in *filename* and fills a VPF_TABLE structure with table information. In doing so, it also loads any variable-length index and identifies the file as to type of standard table (if applicable).

SYNTAX: VPF_TABLE *loadTable(char *filename, int mode)

PARAMETERS:
INPUT:

*filename* – A valid filename of a VPF table including path, if applicable.

*mode* – The mode in which to open the file that must be one of the following integer constants:

*EXISTING* – Read/update – Will not create a new file if the one named is not found.

*READ_ONLY* – Opens the table in a read only mode.

Output: Returns a filled VPF_TABLE type structure containing the file pointer and other information needed to access the table data. Will return NULL if the file cannot be opened, memory cannot be allocated, or the table is in some way invalid.

---

**readDataFile** – Reads an ASCII data file and writes that data into a VPF table. The data must be <CR> delineated ASCII text with any variable-length field preceded by an integer denoting the number of elements in that field.

SYNTAX: int readDataFile(char *filename, VPF_TABLE *table)

PARAMETERS:
INPUT:

*filename* – An ASCII text file containing the data to be entered in the VPF table. Data for all fields in each row you wish to write to the table must appear in this file or the routine will fail.

**table* – A pointer to a VPF_TABLE type structure from the table you wish to enter the data in.

Output: Returns the number of rows written to the table or –1 on error.

---

**readElement** – Reads a typed VPF element from an open file. The file pointer contained in the VPF_TABLE structure passed as parameter one must have read permission and be positioned to the place in the file in which to read the element. A single VPF element may consist of a single value (such as an integer) or a number of values (such as a 3-coordinate array). The function returns the number of *elements* read from file in *count*. The memory returned by this function should be freed when you are through.

SYNTAX: void *readElement(VPF_TABLE *table, int index, int *count)

PARAMETERS:
INPUT:

*table* – A pointer to a VPF_TABLE type structure containing an open file pointer with write permission.

*index* – The column in the VPF_TABLE that contains the format of the element that you wish to read. This corresponds to the type of the element, but is an integer denoting the COLUMN_TYPE structure that holds that information.

*count* – A pointer to an integer that will be filled with the number of elements read. It will usually be one unless the column is a variable-length field.

Output: Returns the number of elements read from file in parameter *count* and a void pointer to an array of those values. Will return NULL on error.

---

**returnData** – Returns an ASCII representation of the data contained in binary form in the parameter *data*. This function does not free the memory that the data passed to it is held in.

SYNTAX: char *returnData(VPF_TABLE *table, void *data, int column, int count)

ARGUMENTS:
INPUT:

*table* – A pointer to a valid VPF_TABLE type structure.

**data* – A void pointer to a VPF data element to be converted.

*column* – The column in the, VPF_TABLE structure's header field that contains the format of the element you wish to convert.

*count* – The number of elements contained in *data.*

Output: Returns a pointer to a NULL terminated character string containing the ASCII representation of the data presented in *data*. On error, NULL is returned.

---

**rewriteTable** – Replaces the rows copied by *copyTable* back to a VPF table. Will read from a temp swap file if *copyTable* was forced to build one. Starts the row numbering from *row* and numbers each successive row accordingly. Does not place the file pointer to the point where the write is to begin; this must be done before calling *rewriteTable.*

SYNTAX: long int rewriteTable(VPF_TABLE *table,

COPY_HEADER *header, long int row);

ARGUMENTS:
INPUT:

*table* – A pointer to a VPF_TABLE type structure from the table you are working on.

*header* – A pointer to a valid COPY_HEADER type structure containing the rows to insert into *table.*

*row* – The number of the first row to be inserted.

Output: Returns the number of the last row written on success or on error it will return a –1.

---

**rowSize** – Returns the length in bytes of the row passed. The variable-length index must have been created if a row is of variable length or the return value will be erroneous. Will return zero if the row requested is out of range of the VPF_TABLE structure passed in *table*.

SYNTAX: long int rowSize(VPF_TABLE *table, long int row)

PARAMETERS:
INPUT:

*table* – A pointer to a VPF_TABLE type structure containing an open file pointer with write permission.

*row* – The number of the row you want the size of.

Output: Returns the length in bytes of the row in question.

---

**setCurrentPosition** – Sets the file pointer of a VPF table to a specified row and column. Either a read-using function *readElement* or a write-using function *writeElement* can be done on return. Returns –1 on error or the byte offset from the beginning of the file on success.

SYNTAX: long int set CurrentPosition(VPF_TABLE *table, long int row, int column)

ARGUMENTS:
INPUT:

*table* – A pointer to a VPF_TABLE type structure from the table you are working on.

*row* – The row of the value you want the file pointer set to.

*column* – The column where the file pointer is to be set.

Output: Returns the byte offset of the field of the row/column position given from the beginning of the file. If the row/column combination is invalid, a -1 is returned and the file pointer is not moved.

---

**splitPath** – Splits the path from a path/filename combination and returns a pointer to a string containing that path. The filename is also returned via *filename*. Any white space is stripped from the beginning and end of both *filename* and *path*. The memory returned by this function should be freed after you are through, using free.

SYNTAX: char *splitPath(char *combination, char **filename)

---

**standardHeader** – Creates a standard VPF table header. These table headers are all taken from the DNC01 database and meet the VPF standard. Returns a pointer to a COLUMN_TYPE array containing the tables' header.

SYNTAX: COLUMN_TYPE *standardHeader(int type, long int *count)

ARGUMENTS:
INPUT:

*type* – An integer constant denoting the type of header to create. A list of the available standard table types and their constants can be found in tools.h.

*count* – A pointer to an integer in which this function will store the number of columns in the header. This information is needed in calls to *buildStandardTable*.

Output: Returns a pointer to a COLUMN_TYPE structure containing a new instance of the table header specified by *type*. On error, NULL will be returned. The number of columns in the returned header will be stored in *count*.

---

**swap_bytes** – Compensates for the variations in the way different architectures store binary data by converting a data structure from most-significant to least-significant byte order (or vice-versa).

SYNTAX: void swap_bytes(unsigned char word[], int num_bytes)

ARGUMENTS:
INPUT:

*word[]* – A contiguously ordered data structure (usually an integer, float, long, etc.).

*num_bytes* – The length of *word[]* in bytes.

Output: Returns nothing directly, but byte-swaps the array *word[]*.

---

**validQuery** – Determines whether a given row/column combination is within a table. Returns YES (1) if valid or NO (0) if not.

SYNTAX: BOOLEAN validQuery(VPF_TABLE *table, long row, int column)

---

**Variable-Length Index Functions**

**addIndex** – Adds an entry to a variable-length index contained in memory as part of the VPF_TABLE structure. Does not update the variable-length index file on disk.

SYNTAX: int addIndex(VPF_TABLE *table, long int length, long int position)

ARGUMENTS:
INPUT:

*table* – A valid VPF_TABLE structure created by the *buildTable* or *loadTable* functions.

*length* – The length in bytes of the row to be entered in the index.

*position* – The row number of the entry.

Output: Returns zero on success or –1 if memory is not available for the entry.

---

**getVarLengthIndex** – Reads and stores the variable-length index associated with the filename stored in the VPF_TABLE structure *table*. Returns a pointer to the head of the variable-length index array read in.

SYNTAX: long *getVarLengthIndex(VPF_TABLE *table)

---

**shrinkIndex** – Deletes an entry in a variable-length index.

SYNTAX: int shrinkIndex(VPF_TABLE *table, long int position)

---

**writeVarIndexTable** – Updates a variable index table on disk.

SYNTAX: int writeVarIndexTable(VPF_TABLE *table)

---

**writeTableDataToFile** – Writes the data from a VPF table into an ASCII file. If a field is a variable length, that entry in the data file will be preceded by an integer denoting the number of elements in the field.

SYNTAX: int write TableDataToFile(char *filename, VPF_TABLE *table)

PARAMETERS:
INPUT:

*filename* – The name of an ASCII text file you wish to write the table data to.

**table* – A pointer to a VPF_TABLE type structure from the table you wish to write out.

Output: Returns the number of rows written to the file or –1 on error.

---

**writeElement** – Writes a typed VPF element to an opened file. The file pointer contained in the VPF_TABLE structure passed as parameter one must have write permission and be positioned to the place in the file in which to write the element. A single VPF element may consist of a single value (such as an integer) or a number of values (such as a 3-coordinate array). The function returns the number of bytes written to file, but if the field is variable length, the number returned may be greater than expected because of the inclusion of field length specifiers in the file.

SYNTAX: long int writeElement(VPF_TABLE *table, int column, int count, void *data)

PARAMETERS:

*table* – A valid VPF_TABLE structure created by the *buildTable* or *loadTable* functions.

*column* – The column in the VPF_TABLE that contains the format of the element that you wish to write. This corresponds to the type of the element, but is an integer denoting the COLUMN_TYPE structure that holds that information.

*count* – An integer denoting the number of elements to be written. It will usually be one unless the column is a variable-length field or a text field.

*data* – A void pointer to an array of values to be written. The number of array positions must be the number of values in the VPF element type count. For instance, if you are writing a variable-length field of type B (a 2-coordinate pair) and length 2 (count = 2), the array passed would be 4 long floating point values.

Output: Returns the number of bytes written to file. Note that if the column is a variable-length field, the byte count may be 4 bytes higher than expected due to the length specifier being inserted into the file.

---

**writeRow** – Writes a row into a VPF table. This function takes a variable number of arguments that make up a row in a VPF table. If the row argument (see below) is out of range of the table, the row is written at the end of the table. The position in the table (rownumber) where the row was written is returned. If the table has an associated variable-length index, that index is updated to reflect the addition of the new row.

SYNTAX: long int writeRow(VPF_TABLE *table, int function, char *fmat, ...)

ARGUMENTS:
INPUT:

*table* – A pointer to a valid VPF_TABLE type structure that has been opened and initialized.

*function* – An integer constant that may be one of the following:

APPEND_ROW – Appends the new row to the end of the table. Ignores the first argument in "...".

REPLACE_ROW – Replaces the row at the position of the first argument in "..." and replaces it with the rest of what's in "...". The existing row is overwritten.

INSERT_ROW – Inserts the row contained in "..." at the position specified in the first argument of "...". None of the rows in the table are lost.

*fmat* – Included for DOS compatibility, it is a pointer to a string containing the format of the first argument in the variable argument list. This string should always be specified as: *char *fmat = %ld;*

... – A variable-length argument list observing the following conventions:

1. The first argument is ALWAYS the row number where you wish the row to be written.
2. The remainder of the arguments are pointers to the data in that row in the order of the columns in the header. In the case of coordinate data types or multiple counts (field lengths), the pointer will be to the first value in an array of values. If a column contains a variable-length field, an integer (not a pointer to an integer) denoting the number of elements in the following argument MUST precede the pointer to the array.

Output: This function returns the number of the row at which the data was written. If the row number given as an argument was in some way incorrect, the row is appended to the table and the row number at which it was written is returned.

---

**writeRowFromList** – Writes the information contained in a ROW_TYPE list into a VPF table. The list presented must be complete or the results may be unpredictable. Any variable-length index associated with the table will be updated to reflect the new row.

SYNTAX: long int writeRowFromList(VPF_TABLE *table, int function, ROW_TYPE *list)

ARGUMENTS:
INPUT:

*table* – A pointer to a valid VPF_TABLE type structure.

*function* – An integer constant defining the action that the function should take. This must be one of the following values:

APPEND_ROW – Writes the row to the end of the table and increments the number of rows by 1. No data are lost.

INSERT_ROW – Writes the row to the row number contained in the first node of the ROW_TYPE structure. In the case of a row number that is out of range of the table, this function reverts to APPEND_ROW. The remaining rows are reordered to accommodate the new row and no data in the table are lost.

REPLACE_ROW – Replaces an existing row with the data contained in the ROW_TYPE structure. The row replaced is the one contained in the first node of the ROW_TYPE structure. In the case of a row number that is out of range of the table, this function reverts to APPEND_ROW.

Output: Returns the number of the row at which the data was written. A –1 is returned if an error occurs.